



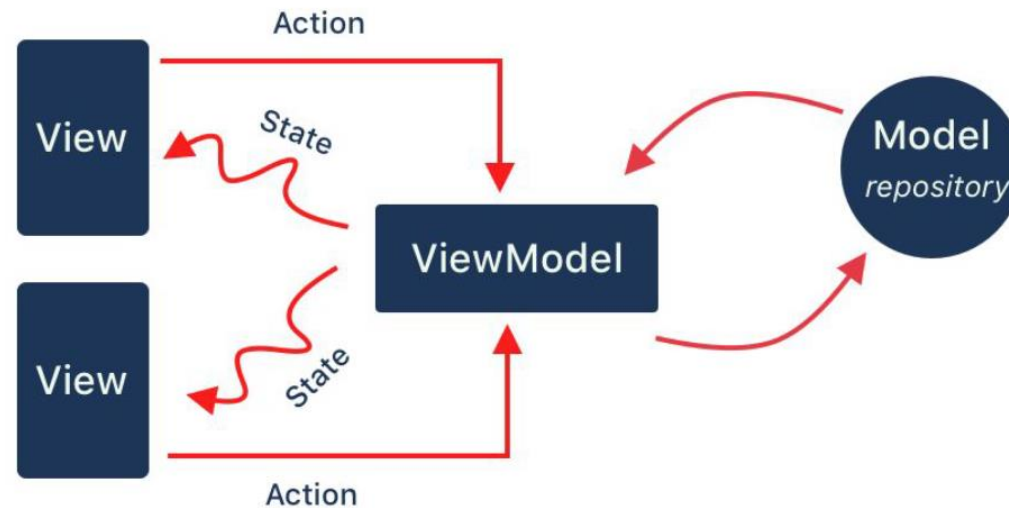
## Model-View-ViewModel (MVVM) Architecture

Dr. Abdelkarim Erradi  
CSE@QU

# Outline

1. Model-View-ViewModel (MVVM)
2. ViewModel
3. LiveData
4. Flow

# MVVM Architecture



# Model-View-ViewModel (MVVM) Architecture

IMPORTANT

**View** = UI to get input from the user.

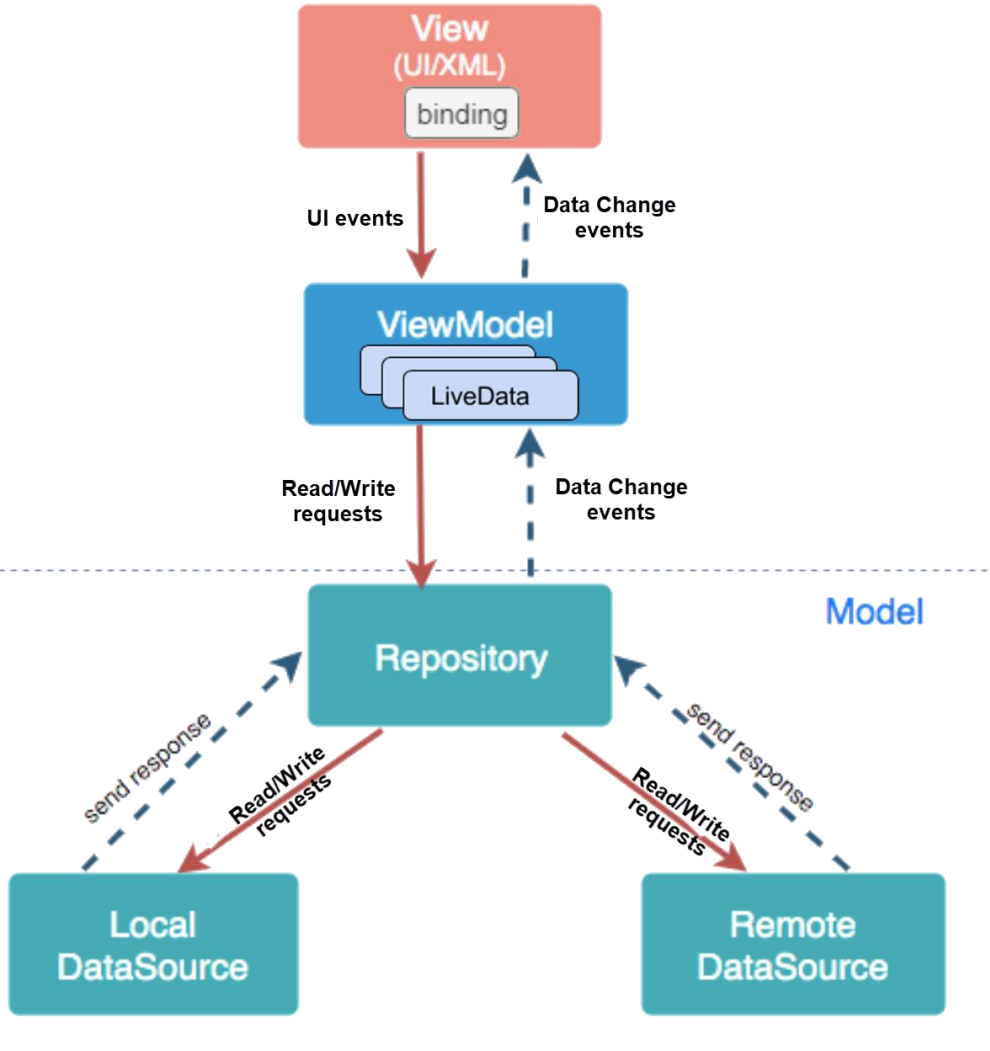
It observes data changes from the ViewModel to update the UI accordingly

## ViewModel

- Holds data needed for the UI
  - Interacts with the Model to read/write data based on user input
  - Notifies the view of data changes
- Implements UI logic / computation

**Model** - handles data operations

- Model has **entities** that represent app data
- Repositories read/write data from either a Local Database (using [Room](#) library) or a Remote Web API (using [Retrofit](#) library)
- Implements data-related logic / computation



# MVVM Key Principles

- Separation of concerns:
  - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
  - ViewModel has no direct reference to the View
  - View never accesses the model directly
  - Model unaware of the view
- Observer pattern:
  - View observes the ViewModel
  - ViewModel observes the Model
- Inversion of Control: *not covered in this course*
  - Uses Dependency Injection instead of direct instantiation of objects


# Advantages of MVVM



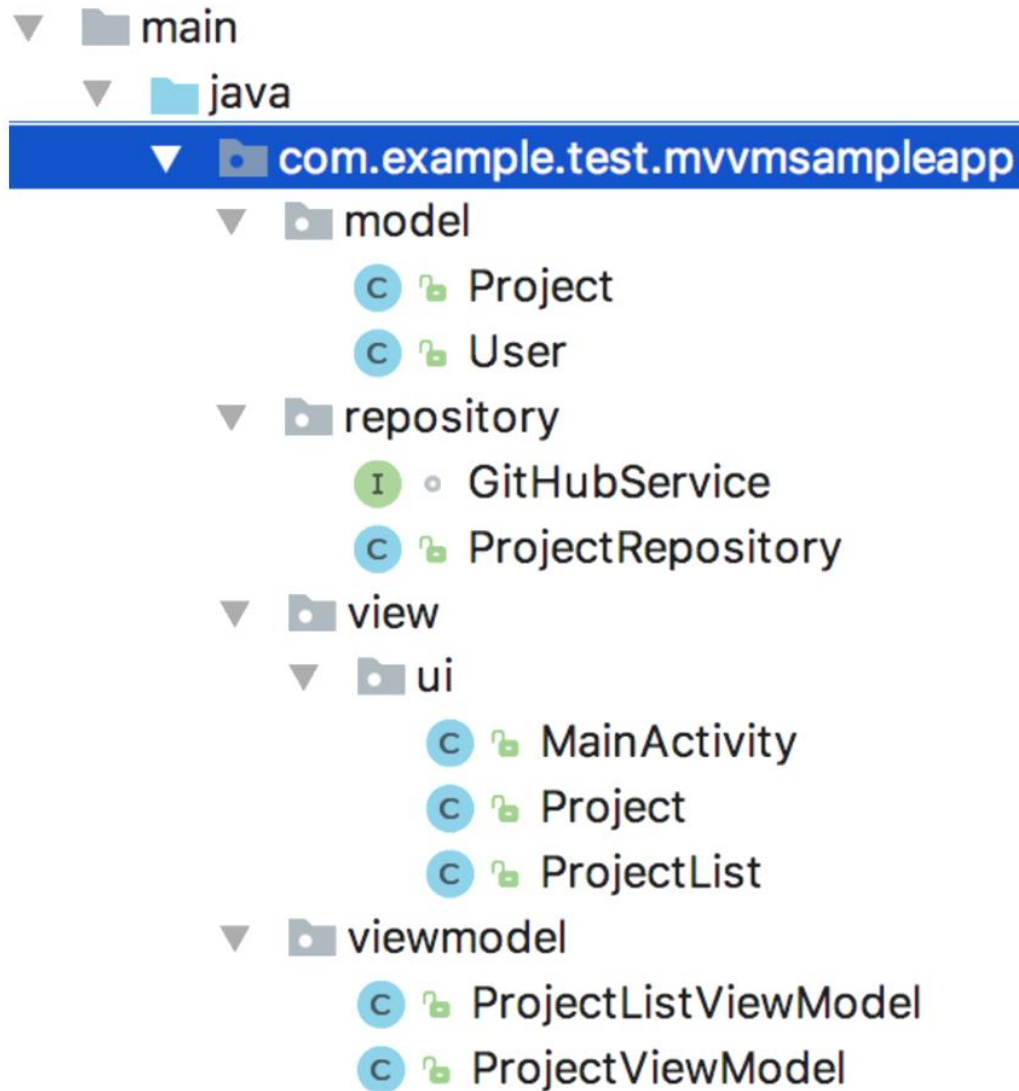
- ***Separation of concerns*** = separate UI from app logic
  - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
  - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
  - Easier **testing** of the App components

MVVM => Easily **maintainable** and **testable** app

# Android Architecture Components

- Android architecture components are a collection of libraries to ease developing MVVM-based Apps
- Part of [Android Jetpack](#)  They include:
  - [ViewModel](#) stores UI-related data that isn't destroyed on screen rotation
  - [LiveData](#) data holder that notifies the View when the underlying data changes
  - [Room](#) to read / write data to local SQLite database

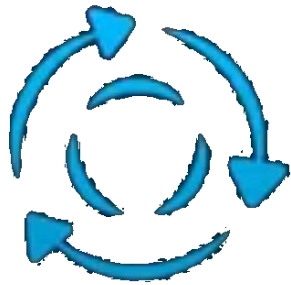
# Recommended Project Structure



You may  
organize the  
view by feature



# ViewModel



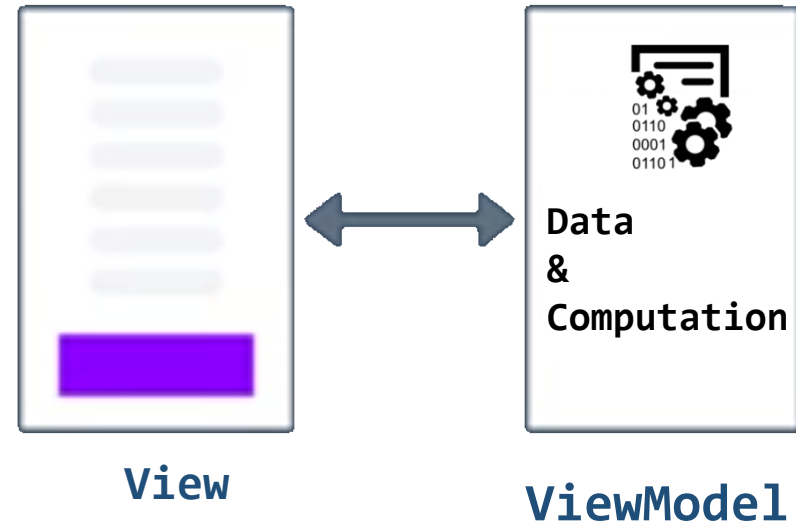
**Lifecycle Aware**



**Survives Config Changes**

# ViewModel

- ViewModel is used to **store and manage UI-related data**
  - in a lifecycle conscious way
  - allows data to survive device configuration changes such as *screen rotations* or *changing the device's language*
- If the system destroys or re-creates a UI component (e.g., when the screen rotates), any transient UI-related data you store in it is lost



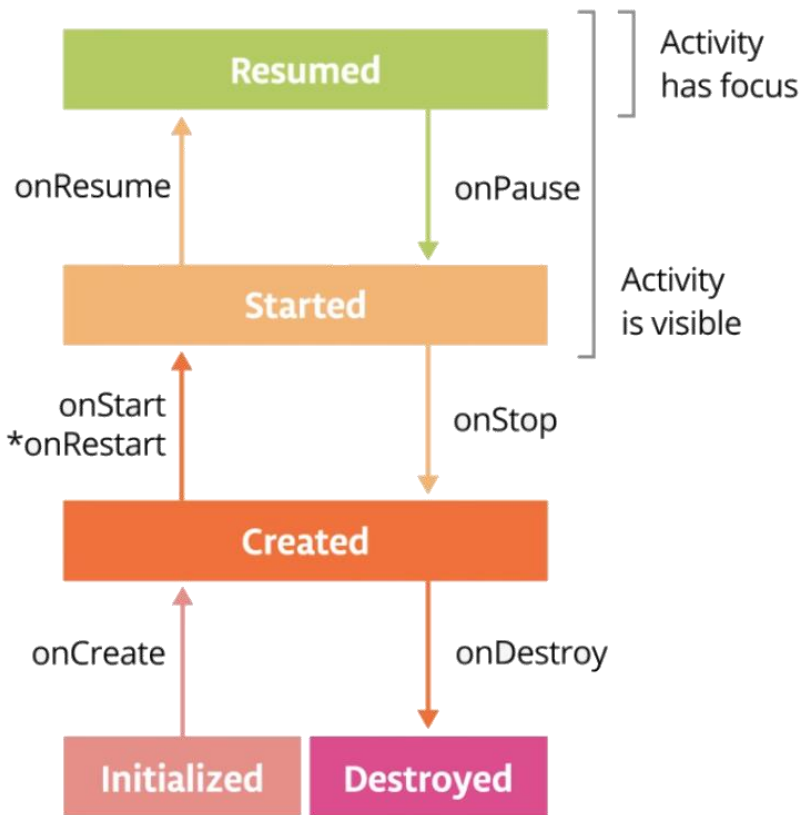
## User **ViewModel**:

- Store UI data
- Read/write data from a Repository

# Activity Lifecycle

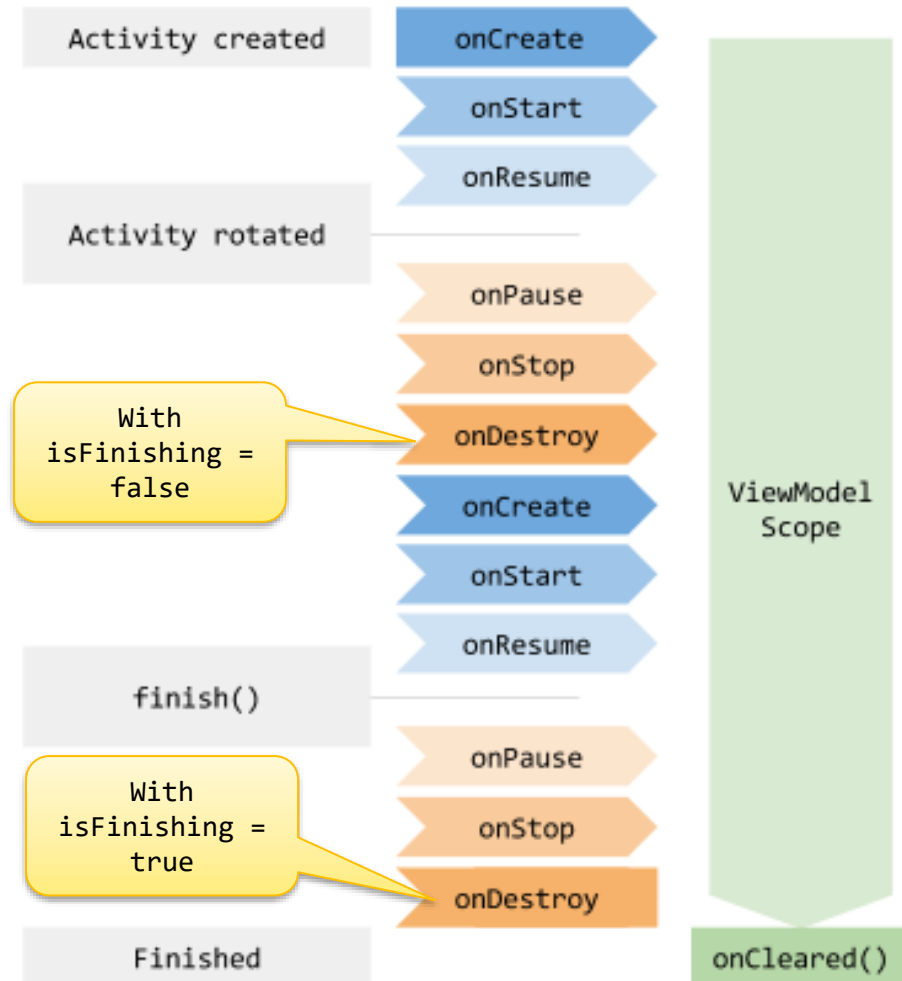
An activity has essentially **four** states:

- **Resumed** if the activity is in the foreground of the screen (has focus)
- **Started** if the activity has lost focus but is still visible (e.g., beneath a dialog box).
  - When the user returns to the activity, it is **resumed**
- **Created** if the activity is completely obscured by another activity.
  - When the user navigates to the activity, it must be **restarted** and restored to its previous state.
- **Destroyed** when the user closes the app or if the activity is killed (when memory is needed or due to **finish()** being called on the activity)



# ViewModel Lifecycle

- ViewModel object can be scoped to the main activity
- However, it has a **longer lifespan** compared to the associated Activity which may undergo a rotation and get recreated
- It remains in memory until the activity is completely destroyed
  - When the activity is recreated (after a screen rotation) the associated ViewModel remains alive



# ViewModel Example

```
class ScoreViewModel : ViewModel() {  
    var team1Score = 0  
    fun incrementTeam1Score() = team1Score++  
}
```

*/\* To allow any screen to get an instance of the **shared** viewModel  
make the viewModel **scoped** to the activity (by assigning the activity as  
the viewModel Store Owner)  
This ensures that the same viewModel instance is used for all screens  
\*/*

```
val scoreViewModel = viewModel<ScoreViewModel>  
    (viewModelStoreOwner = LocalContext.current as ComponentActivity)  
val team1Score = scoreViewModel.team1Score
```

Need to add to build.gradle:

```
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.4.0-beta01"
```

# Associate ViewModel with the Activity

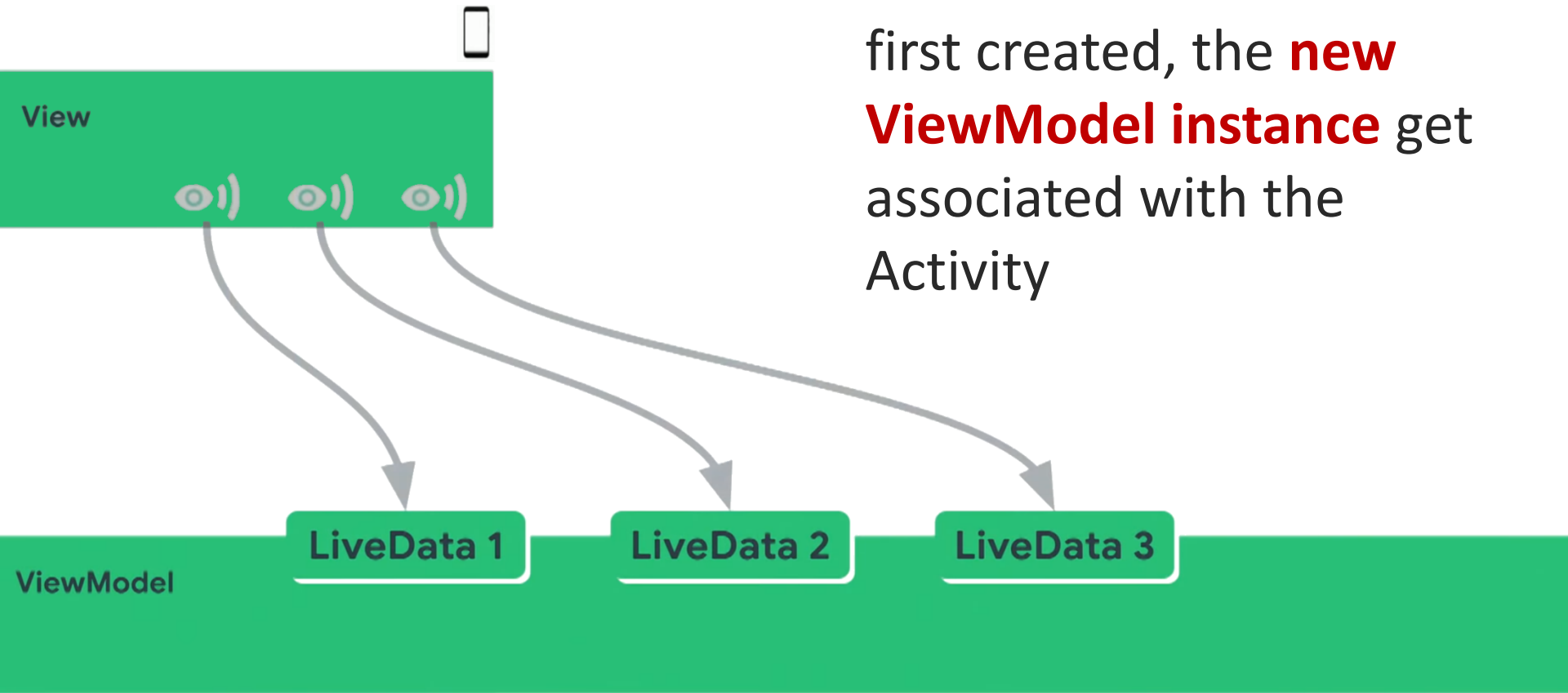
- Use **viewModel()** method to get an instance of the ViewModel

```
val scoreViewModel = viewModel<ScoreViewModel>  
    (viewModelStoreOwner = LocalContext.current as ComponentActivity)
```

- For the first call, this creates and returns a new ViewModel instance **scoped** to the activity (i.e., associated with the Main Activity) to make it **shared** and accessible from all screens
- For subsequent calls, **viewModel()** method will return the pre-existing ViewModel scoped to the Activity

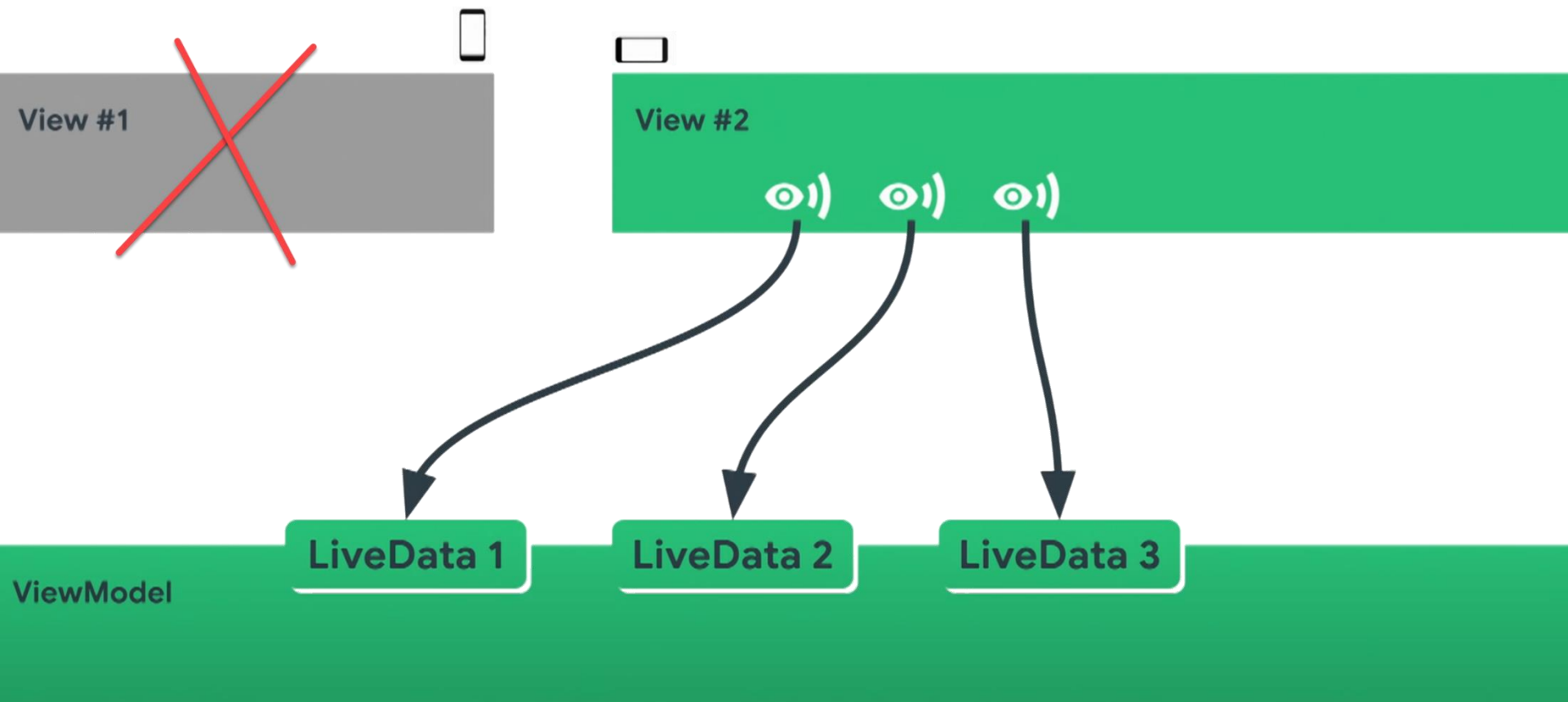
# When the ViewModel is first Created

When the ViewModel is first created, the **new ViewModel instance** get associated with the Activity



# OnConfig change (e.g., Screen Rotates)

OnConfig change, the Activity is destroyed, and a new instance of the Activity is created then it obtains **the same ViewModel instance used previously**





# “no contexts in ViewModels” rule

- ViewModel should **not be aware of the View** who is interacting with  
=> It should be **decoupled** from the View

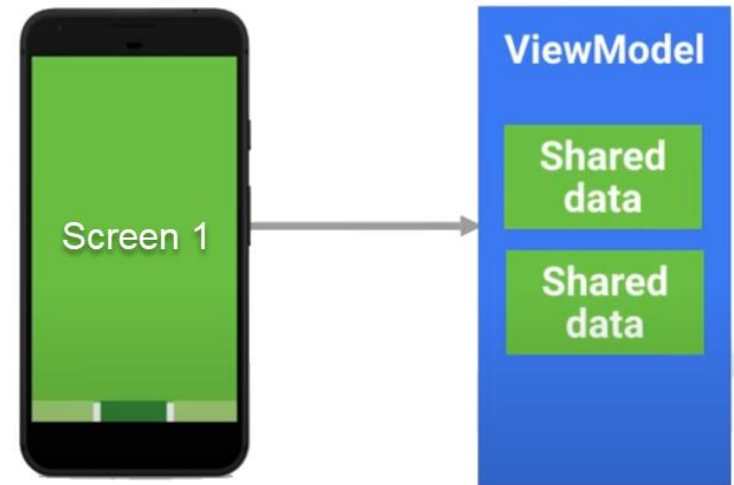


- ViewModel should not hold a reference to Activities or Views (i.e. Composables)
- Should not have any Android framework related code
- As this defeats the purpose of separating the UI from the data
- Can lead to **memory leaks** and **crashes** (due to null pointer exceptions) as the ViewModel outlives the View
  - if you rotate an Activity 3 times, 3 three different Activity instances will be created, but you only have one ViewModel instance

# Shared data between Screens using ViewModel



- Screens can **share** data using a shared **View Model** class that extends `ViewModel()`



`@Composable`

```
fun ProfileScreen(userId: Int) {  
    /* Get an instance of the shared viewModel  
       Make the activity the store owner of the viewModel  
       to ensure that the same viewModel instance is used for all screens */  
    val userViewModel = viewModel<UserViewModel>(viewModelStoreOwner =  
        LocalContext.current as ComponentActivity)  
    val user = userViewModel.getUser(userId)  
    ... }  
}
```

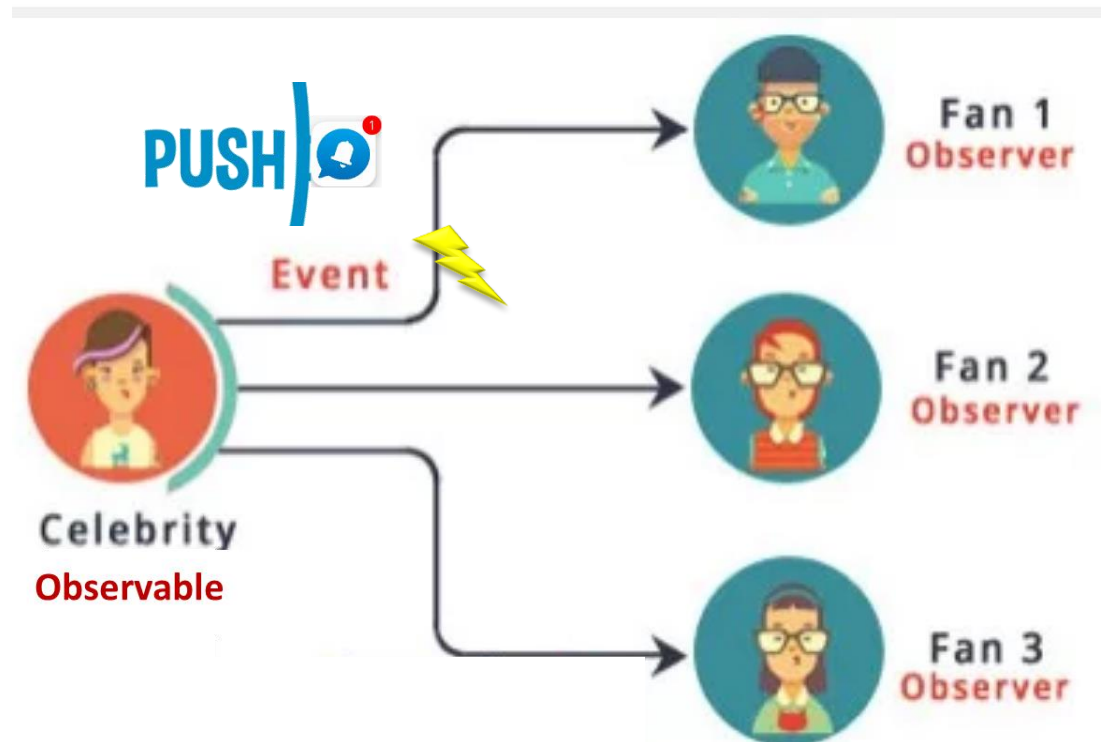
# LiveData

# LiveData

- LiveData is an **observable data holder**: **active** observers (i.e., the View) get notified when data change
- The view can observe LiveData objects for changes without creating **explicit and rigid dependency** between them
  - This decouples completely the LiveData object producer from the LiveData object consumer
  - ViewModel exposes its data using **LiveData** that the View can observe and update the UI accordingly

# Observable - Real-Life Example

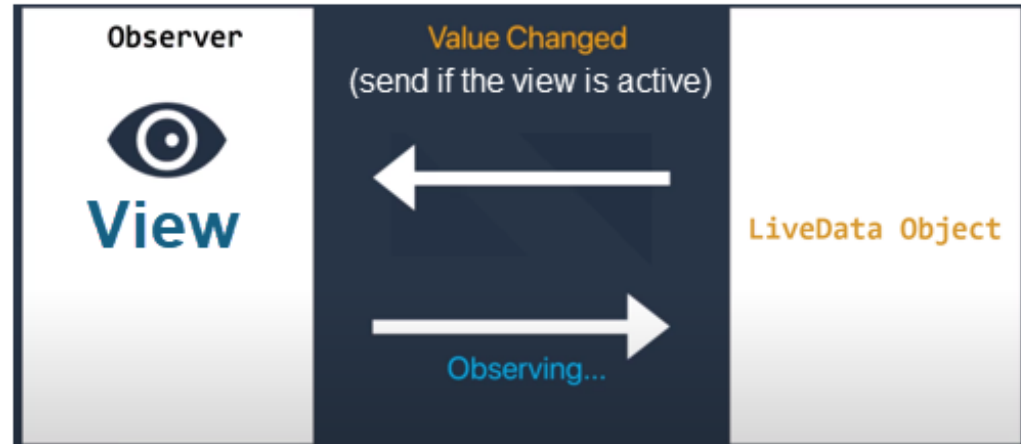
- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable** (called **LiveData** on Android)



# LiveData is lifecycle-aware

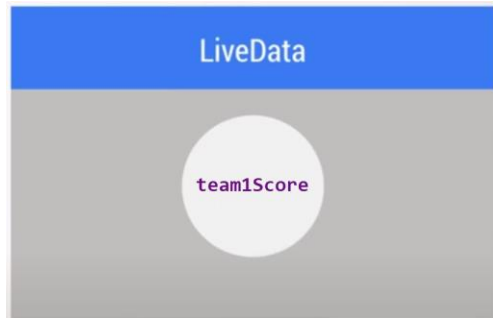
## LiveData is aware of the Lifecycle of its Observer

- Notifies data changes to only **active** observers (Stopped/Destroyed View will NOT receive updates)
- It automatically removes the subscription when the observer is destroyed so it will not get any updates



# LiveData in Code

LiveData **warps around** an object and allows the view to **observe** it



↑  
Observes



- ViewModel expose LiveData objects that the View can observe

```
class MainActivityViewModel : ViewModel() {  
    private val _team1Score = MutableLiveData<Int>(0)  
  
    // Expose read only LiveData that the View can observe or bind to  
    val team1Score: LiveData<Int> get() = _team1Score  
  
    fun incrementTeam1Score() {  
        // call postValue to notify Observers  
        _team1Score.value = (_team1Score.value ?: 0) + 1  
    }  
}
```

- View **observes** LiveData changes

```
class MainActivity : AppCompatActivity() {  
    // onCreate  
    // Associate the Activity with the ViewModel  
    val viewModel by viewModels<MainActivityViewModel>()  
  
    viewModel.team1Score.observe(this) {  
        team1ScoreTv.text = it.toString()  
    }  
}
```

# LiveData.observeAsState()

- LiveData.**observeAsState()** registers as a **listener** and represent the values as a State.
  - Whenever a new value is emitted, Compose recomposes those parts of the UI where that state.value is used
  - Required dependency:  
androidx.compose.runtime:runtime-livedata:\$composeVersion



## Example - LiveData.observeAsState()

- For example, in this code, **ShowData** recomposes every time **exampleLiveData** emits a new value


@Composable

```
fun MyExample() {  
    val viewModel: ExampleViewModel = viewModel()  
    val dataExample = viewModel.exampleLiveData.observeAsState()  
  
    // Because the state is read here,  
    // MyExample recomposes whenever dataExample changes  
    dataExample.value?.let {  
        ShowData(dataExample)  
    }  
}
```

# Flow




# What is Flow?

 Stream of values (produced one at a time instead of all at once)

 Values could be generated from network requests or database calls

 Can transform a flow using operators like map, switchMap, etc

```
fun stream(): Flow<String> = flow {  
    emit("🦖") // Emits the value upstream   
    emit("🎲")  
    emit("🍷")  
}
```



# Return Flow: Stream of Data



```
object WeatherRepository {  
    private val weatherConditions = listOf("Sunny", "Windy", "Rainy", "Snowy")  
    fun fetchWeatherFlow(): Flow<String> =  
        flow {  
            var counter = 0  
            while (true) {  
                counter++  
                delay(2800)  
                emit(weatherConditions[counter % weatherConditions.size])  
            }  
        }  
}
```

```
val currentWeatherFlow: LiveData<String> =  
    WeatherRepository.fetchWeatherFlow().asLiveData()
```

# Flow Operators

- Flow has operators similar to collections such as map, filter and reduce

```
(1..5).asFlow()  
    .filter { it % 2 == 0 }  
    .map { it * it }  
    .collect { println(it.toString()) }
```

```
val result =(1..5).asFlow()  
                .reduce { a, b -> a + b }  
println("result: $result")
```

# Resources

- MVVM
  - <https://developer.android.com/jetpack/guide>
  - <https://medium.com/androiddevelopers/viewmodels-a-simple-example-ed5ac416317e>
- Data Binding
  - <https://developer.android.com/topic/libraries/data-binding>
- Data Binding codelab
  - <https://codelabs.developers.google.com/codelabs/android-databinding>